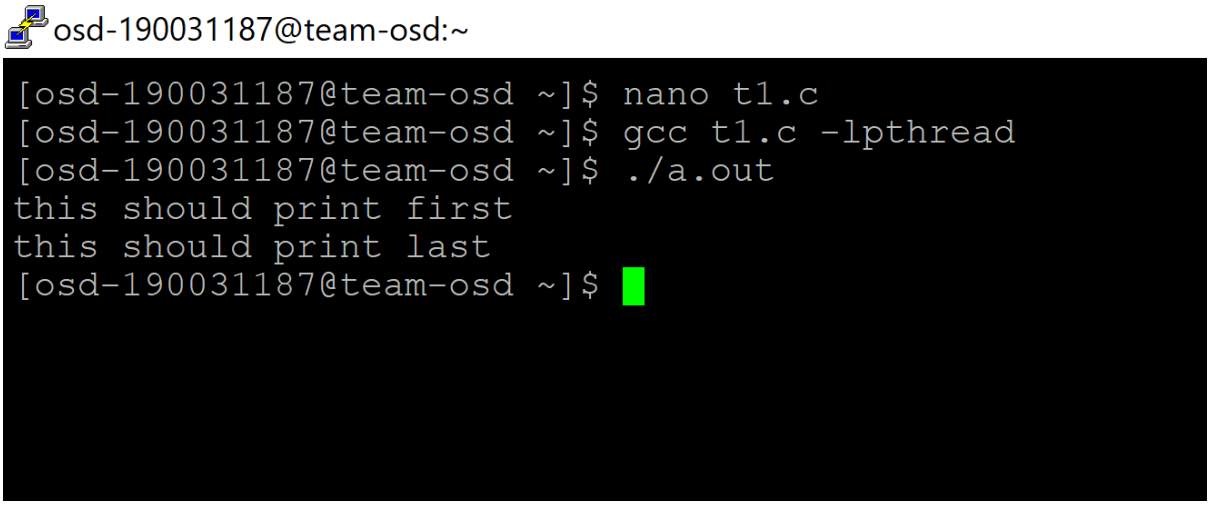
Operating System and Design (19CS2106S)
Lab- 11
Pre-Lab

1. Creating and Destroying Mutexes: pthread_mutex_init (mutex,attr) pthread_mutex_destroy (mutex)

```
#include <stdio.h>
#include <pthread.h>
int done = 0;
void* worker(void* arg) {
   printf("this should print first\n");
   done = 1;
   return NULL;
}

int main(int argc, char *argv[]) {
   pthread_t p;
   pthread_create(&p, NULL, worker, NULL);
   while (done == 0)
         ;
   printf("this should print last\n");
   return 0;
}
```

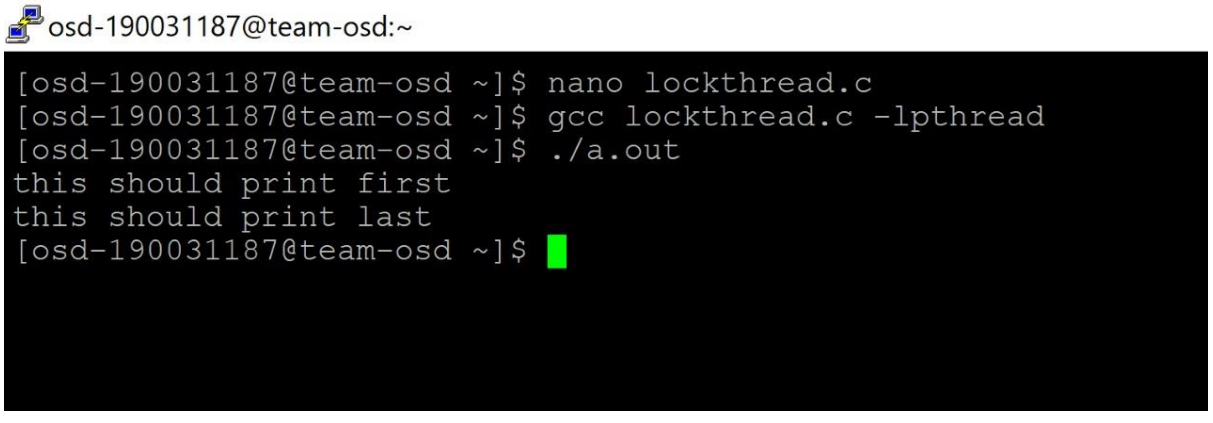


2. Locking and Unlocking Mutexes: pthread_mutex_lock (mutex) pthread_mutex_trylock (mutex)

```
#include <stdio.h>
#include <pthread.h>

typedef struct __synchronizer_t {
   pthread_mutex_t lock;
   pthread_cond_t cond;
```

```
   int done;
} synchronizer_t;
synchronizer_t s;
void signal_init(synchronizer_t *s) {
   pthread_mutex_init(&s->lock, NULL);
   pthread_cond_init(&s->cond, NULL);
   s->done = 0;
}
void signal_done(synchronizer_t *s) {
   pthread_mutex_lock(&s->lock);
   s->done = 1;
   pthread_cond_signal(&s->cond);
   pthread_mutex_unlock(&s->lock);
}
void signal_wait(synchronizer_t *s) {
   pthread_mutex_lock(&s->lock);
   while (s->done == 0)
         pthread_cond_wait(&s->cond, &s->lock);
   pthread_mutex_unlock(&s->lock);
}
void* worker(void* arg) {
   printf("this should print first\n");
   signal_done(&s);
   return NULL;
}
int main(int argc, char *argv[]) {
   pthread_t p;
   signal_init(&s);
   pthread_create(&p, NULL, worker, NULL);
   signal_wait(&s);
   printf("this should print last\n");
   return 0;
}
```



3. **Synchronizing Threads with POSIX Semaphores: sem_init, sem_open sem_wait sem_post sem_getvalue sem_destroy**

Semaphores:

Semaphores are used for process and thread synchronization. Semaphores are clubbed with message queues and shared memory under the Interprocess Communication (IPC) facilities in Unix-like systems such as Linux.

sem_init

```
#include <semaphore.h>
int sem_init (sem_t *sem, int pshared, unsigned int value);
```

sem_init is the equivalent of sem_open for unnamed semaphores. One defines a variable of type sem_t and passes its pointer as *sem* in the sem_init call.

sem_open

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

sem_t *sem_open (const char *name, int oflag);
sem_t *sem_open (const char *name, int oflag,
        mode_t mode, unsigned int value);
```

sem_open is the call to get started for a semaphore. sem_open opens an existing semaphore or creates a new semaphore and opens it for further operations.

sem_post

```
#include <semaphore.h>
int sem_post (sem_t *sem);
```

sem_post increments the semaphore. It provides the V operation for the semaphore. It returns 0 on success and -1 on error.

sem_wait

```
#include <semaphore.h>
int sem_wait (sem_t *sem);
```

sem_wait decrements the semaphore pointed by *sem*. If the semaphore value is non-zero, the decrement happens right away. If the semaphore value is zero, the call blocks till the time semaphore becomes greater than zero and the decrement is done. sem_wait returns zero on success and -1 on error.

sem_getvalue

```
#include <semaphore.h>
int sem_getvalue (sem_t *sem, int *sval);
```

sem_getvalue gets the value of semaphore pointed by sem. The value is returned in the integer pointed by sval. It returns 0 on success and -1 on error, with errno indicating the actual error.

sem_destory

```
#include <semaphore.h>
int sem_destroy (sem_t *sem);
```

sem_destroy destroys the unnamed semaphore pointed by sem.

In-Lab

1. **Illustrate how mutex is used for thread synchronization, print the counter variable upon each increment which is in the critical section. (Two threads update a global shared variable with and without synchronization).**

```
#include <pthread.h>

static volatile int glob = 0;   /* "volatile" prevents compiler optimizations
                                   of arithmetic operations on 'glob' */
static void *                /* Loop 'arg' times incrementing 'glob' */
threadFunc(void *arg)
{
   int loops = *((int *) arg);
   int loc, j;

   for (j = 0; j < loops; j++) {
      loc = glob;
      loc++;
      glob = loc;
   }
   return NULL;
}
int
main(int argc, char *argv[])
{
   pthread_t t1, t2;
   int loops, s;
   loops = strtol(argv[1], NULL, 10);
   s = pthread_create(&t1, NULL, threadFunc, &loops);
   if (s != 0){
      perror("pthread_create fail");
          exit(1);}
   s = pthread_create(&t2, NULL, threadFunc, &loops);
   if (s != 0){
      perror("pthread_create fail");
          exit(2);}

   s = pthread_join(t1, NULL);
   if (s != 0){
      perror("pthread_join fail");
          exit(1);}
   s = pthread_join(t2, NULL);
   if (s != 0){
      perror("pthread_join fail");
          exit(1);}

   printf("glob = %d\n", glob);
```

```
   exit(0);
}
```

osd-190031187@team-osd:~

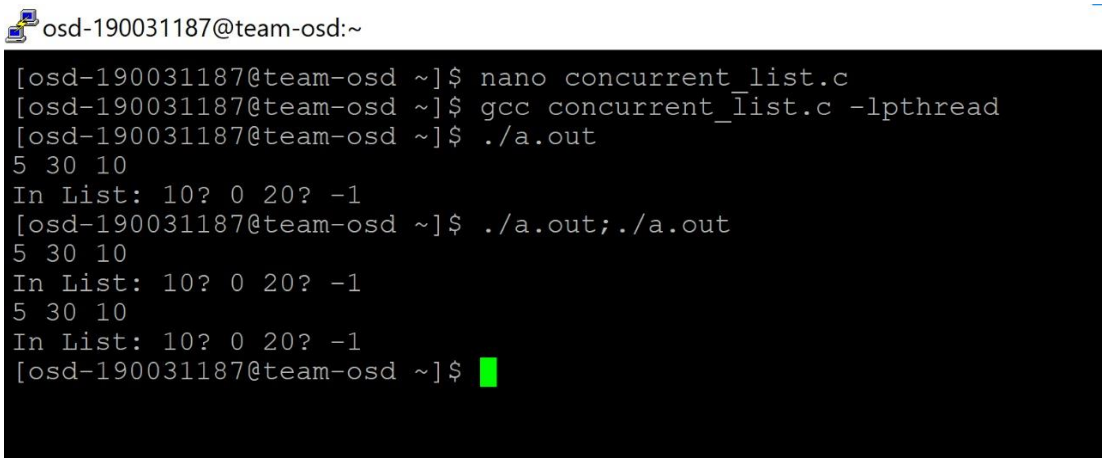
```
[osd-190031187@team-osd ~]$ nano thread_mutex.c
[osd-190031187@team-osd ~]$ gcc thread_mutex.c -lpthread
[osd-190031187@team-osd ~]$ ./a.out 500
glob = 1000
[osd-190031187@team-osd ~]$ ./a.out 5000
glob = 10000
[osd-190031187@team-osd ~]$ ./a.out 5000000
glob = 10000000
[osd-190031187@team-osd ~]$
```

2. Write a UNIX system program to implement concurrent Linked List

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
typedef struct __node_t {
   int             key;
   struct __node_t      *next;
} node_t;
// basic list structure (one used per list)
typedef struct __list_t {
node_t *head;
pthread_mutex_t lock;
} list_t;
        void List_Init(list_t *L) {
                L->head = NULL;
                pthread_mutex_init(&L->lock, NULL);
        }
        void List_Insert(list_t *L, int key) {
                // synchronization not needed
                node_t *new = malloc(sizeof(node_t));
                if (new == NULL) {
                        perror("malloc");
                        return;
                }
                new->key = key;

                // just lock critical section
                pthread_mutex_lock(&L->lock);
                new->next = L->head;
                L->head = new;
```

```
                pthread_mutex_unlock(&L->lock);
        }
        int List_Lookup(list_t *L, int key) {
                int rv = -1;
                pthread_mutex_lock(&L->lock);
                node_t *curr = L->head;
                while (curr) {
                        if (curr->key == key) {
                                rv = 0;
                                break;
                        }
                        curr = curr->next;
                }
                pthread_mutex_unlock(&L->lock);
                return rv; // now both success and failure
        }
void List_Print(list_t *L) {
   node_t *tmp = L->head;
   while (tmp) {
        printf("%d ", tmp->key);
        tmp = tmp->next;
   }
   printf("\n");
}
int main(int argc, char *argv[])
{
   list_t mylist;
   List_Init(&mylist);
   List_Insert(&mylist, 10);
   List_Insert(&mylist, 30);
   List_Insert(&mylist, 5);
   List_Print(&mylist);
   printf("In List: 10? %d 20? %d\n",
          List_Lookup(&mylist, 10), List_Lookup(&mylist, 20));
   return 0;
}
```



Post-Lab

1. **Write a Unix System program to make A Parent Waiting for Its Child using semaphores.**

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
sem_t done;
void *
child(void *arg) {
   sleep(5);
   printf("child\n");
   sem_post(&done);
   return NULL;
}
int
main(int argc, char *argv[]) {
   pthread_t p;
   printf("parent: begin\n");
   sem_init(&done,0, 0);
   pthread_create(&p, NULL, child, NULL);
   sem_wait(&done);
   printf("parent: end\n");
   return 0;
}
```